Grph: an efficient portable graph library tailored to network simulation and graph analysis

July 17, 2012

Contents

Abstract

1 Introduction

1.1 General description

Many scientific and technical fields relate to graphs. Researchers and engineers who need to programmatically manipulate graph structures have access to a number of existing graph frameworks [5, 6, 2, 3]. Unfortunately, in spite of the intrinsic qualities of each specific framework, their all turn out to have strong limitations in either their execution platform, their computational performance, the flexibility of their graph model, etc. As a consequence scientists most often opt for developing ad hoc tools. The very same code is then invariably re-written by different groups of people, providing the same functionality and exhibiting the same mistakes. These mistakes include inappropriate design and implementation techniques which lead to poor performance and thereby the unability to deal with large graphs.

Grph is a Java graph toolkit enabling experimentation on large graphs (in the order of millions of nodes). It proposes a general graph models which supports directed and undirected simple and hyper-edges in a mixed fashion. Its design and implementation objectives are geared towards:

- computational efficiency;
- memory efficiency;
- simplicity.

To this purpose, *Grph* exhibits the following features:

- a data model which represents vertices and edges as positive integers;
- a efficient data structure based on incidence lists;
- implementations for most common graph algorithms;
- a framework for the development of new algorithms and their integration with *Grph*;
- a graphical monitoring interface;
- a console-based interactive interface (shell).

1.2 State of the Art

Among the numerous graph software which already are available to the graph community, projects include Mascopt [4], GraphStream [1], Boost [6], LEDA, JGraphT [2], JUNG [5], SAGEMath [3], etc.

This document focuses on Java software, hence it does not consider greats tools like Boost or LEDA, which are in C++ or SAGEMath which is implemented in Python, with some slices of C.

Java users can choose in a plethora of frameworks. Most of them are lab tools which still are at an early stage of their development and would not be used in a production environment. Mature graph projects include JUNG, Apache Maths Commons and GraphT stand among the most often used. Unfortunately most of these frameworks suffer from a lack of support for certain class of graphs (mixed graphs, hypergraphs), from an often complex design, and from poor performance. In particular their full-OO design (which allows the construction `Graph<V, E>` when V and E can be of any type) imposes their implementation to rely on hashtables. The immediate consequence of using hashtables in Java is the large memory usage and slow access time (even is hashtables theoretically operate in constant time, they prove much slower than arrays).

1.3 Motivation

Our initial motivation for the *Grph* project was to build up a graph toolkit that would help us in the frame of a Research project at INRIA labs, whose the problem was the simulation of large networks. Later we realized the potential of our graph toolkit to compete with the best ones. Then our motivation changed: our aim is now to deliver that can help anyone researcher or engineers in need of graph-related functionality, wether it be in academical research of industrial context.

1.4 Why calling it "*Grph*"?

Why have chosen *Grph* as the library name. The toolkit was initially called *Dipergrafs* because it focused on the manipulation of directed hypergraphs. Since the objectives have somehow changed upon times, the name had to be adapted to. Now the target model is general graphs mixed in terms of the nature of the edges (simple and hyper) and in their directivity (directed or not). We opted for "Grph" because the missing 'a' retains lots of attention and it is short enough so it can be used within the API.

2 Graph model supported

2.1 Topology models

Historically, the initial motivation for developing *Grph* was to make it useful in the context of network simulation, considering that networks involve a number of different technologies that must be modeled in their own special manner: Ethernet bus, optical links, Wi-Fi/Bluetooth wireless networks, duxplex/full duplex communication channels, etc. Undirected/directed simple/hyper edges provide the adequate abstractions to represent this technological variety.

Then *Grph* comes with a general graph model that supports:

- simple graphs;
- multigraphs;
- hypergraphs;
- any undirected, directed or mixed versions of these (directed hypergraphs are still under development);
- mixed graphs.

2.2 Data model

2.2.1 Vertices and edges are integers

Most graph toolkits provide models for graphs of *objects*: they allow vertices and objects to be of any type. This elegant OO design has a rude counterpart: it prevents good performance because it requires:

- the computation of hashcodes;
- the use of hashtables, which have heavy memory footprint, and exhibit slow operation;
- the use of pointers, which require 64 bits on modern computers and add indirections;
- the user of objects, which have large (at least 12 bytes) overhead.

Grph follows another approach, geared toward performance: it models vertices and edges as numbers. These number are stored in contiguous sequences of native integer values, which removes the need to hashcode, hashtables, pointers and objects.

The spaces of identifiers do not have to be contiguous, though the data structure exhibit de best memory performance when these spaces are dense.

2.3 Properties

As said previously, vertices and edges in *Grph* are not object but numbers. Because of this, the properties for vertices/edges cannot be defined by as attributes of the the vertex and edge classes.

Grph define a number of basic properties for vertices and edges:

label (string);

color (4 bits);

size (4 bits);

style (1 bit);

These properties all have an obvious graphically representation, and are actually used in the monitoring view and Graphviz/DOT export method.

These default properties may be used as building blocks for application level properties. For example, if the users needs a *rank* property for vertices, he may want to define a new data structure for the storage of the rank values and define the corresponding getter/setter methods. A cheaper solution consists as re-using the *size* vertex property. Getter/setter methods for the *rank* property would then simply make direct calls to the getter/setter methods for the *size* property. By using this solution, altering the rank would directly be observable by a modification of the size of the corresponding vertex on the graphical view.

2.3.1 Pro and cons of storing properties in the graph

The main disadvantage of this model for accessing and storing properties is that it is not a usual approach of people used to work in Java OO world.

But the advantages are:

- improved memory usage;
- automatic observation of property values;
- automatic search/indexing

2.3.2 Searching properties

The property framework *Grph* features a function which returns a set of elements that have a given value for a given property. This method is

```
IDSet IntProperty.findElements(int value)
```

This function iterates on the set of elements by storing elements that are found to have the requested value. If the performance of this search function is critical to your application, you may want to index the value of properties for set of elements that store it. This strategy is commonly used in database systems where is it referred to as *cursor*. A property can be asked to use a cursor via its `setUseCursor(boolean)` method. The complexity of the search operation is then:

- $\Theta(1)$ if the cursor is used;
- $\Theta(n)$ otherwise.

Using a cursor improves the search of elements when the number of element is large but it significantly slows down the operations:

- changing the property value for elements;
- adding/removing elements from the graph.

2.3.3 Managing graphs of objects

Although this model matches well the mathematical concept of a graph and is adequate to developments that come atop *Grph*, many situation still require the vertices and/or edges to be modelled as objects (for the example when the data model was defined prior to the decision of using *Grph*).

Grph solves this issue by providing a bridge from the object world to the integer world, and the other way around. The class `grph.ObjectGrph<V, E>` encapsulates an instance of an integer-based graph and provide the necessary conversion methods: from IDs to objects (vertices or edges) and from set of IDs to sets of objects.

Because of the number of API methods *Grph* available to the developer is large, the class `grph.ObjectGrph<V, E>` cannot reasonably bridge all of them. *Grph* considers that the conversion methods provided will serve the user at implementing the object-based methods he needs, by bridging the already integer-based methods available.

3 Data structure

Grph comes with one single implementation for the graph data structure which is based on four coupled incidence lists. It is commonly admitted that incidence list based implementation perform well when graphs are sparse and that dense graphs are more adequately represented by matrices. In practice, people dealing with dense graphs are theory animals who seldom opt for programmatic solutions anyway.

The incidence lists used by *Grph* store the IDs of incident elements. They are indexed by the ID of element they relate to. The cell at index n in the vertex incidence list contains the set of ID of the edges incident to vertex n. This cell is left empty if vertex n is not in the graph.

3.1 Element sets

Many functions returns set of ID (either vertex IDs or edge IDs). For the sake of performance *Grph* may return references to ID sets that are used internally, instead of returning copies. Altering the content for these set would compromise the integrity of graphs. To this purpose, the returned sets are flagged, indicated weather they are internal sets.

When dealing with a given set obtained via the use of a *Grph* method, it is possible to make sure that the set is not an internal set by calling the `ensureCopy()` method.

3.2 Property value

The way the basic property values are stored is up to the developer. Within *Grph*, default properties are stored in dedicated arrays which allow small memory usage.

In order to capture the benefit of using this strategy, consider as an example a *marked* vertex property that can take two values. The smallest data types in Java that can be used to store a rank value are boolean or bytes (both are encoded on 8 bits).

- if vertices were modelled as objects, every vertex would occupy 192 bits of memory: 12 bytes of Java object overhead) + 1 byte for the mark (but actual size of the object would be 16 bytes because the size for any java object must be multiple of 8) + 64-bit reference to the vertex;
- in the case vertices are modelled as numbers, the space required to store one vertex and its *marked* is 33 bits: 32 bit integer for the vertex ID + 1 bit for the boolean value (that can be stored in a bitset storing all *marked* values for all vertices).

This example, which represents a very common situation, shows a memory usage that is 5.8 times lower when using integer vertices.

3.3 Computational complexity

This model exhibit the following computational complexity:

number of elements is the number of non-empty cells in the list; since the number of empty cells is stored and updated on-the-fly, computing the number of elements simply consists in substracting the number of empty cells to the actual size of the incidence list; this is done in all case $0(1)$;

looking up an element is done by checking if a cell a the given index is empty or not; this is done in $0(1)$

adding an element is done by setting a cell at the index given by the ID of the the element with an instance of an object storing the IDs of its incident elements; this is done in $0(1)$;

removing an element is done by emptying a cell at a given index and updating the incident set of its incident elements; this is done in $0(n)$, n being the number of elements incident to the element to be removed — its degree;

iterating over the elements in the set is done by sequentially iterating over the list, ignoring empty cells; this is done in $0(n+m)$, n being the number of elements of the corresponding type in the graph and m the number of empty cells in adjacency list. Most often m tends to zero.

Compared performance shows that using HPPC for manipulation vertices/edges is 7 time faster than using Java collections when it comes to creating them, and 2.5 times faster when it comes to iterating over them.

3.4 Memory complexity

The incidence of every element (vertex or edge) is represented by an object.

- a simple edge, either directed or not, takes 20 bytes: 4 (ID)+16 (incident vertices)
- an undirected hyperedge takes $4 + 16 + 12 + n \times 4$, n being the number of incident vertices.
- a directed hyperedge takes $4 + 24 + n \times 4$, n being the number of incident vertices.
- a vertex takes 4+24 bytes.
  - adds $12 + n \times 4$ if it has n out directed edges;
  - adds $12 + n \times 4$ if it has n in directed edges;
  - adds $12 + n \times 4$ if it has n undirected edges;

A graph with n vertices and m edges takes $24 + n(40 + 4\frac{m}{n}) + 20m$ bytes.

4 Algorithms

Grph comes with implementations for most common graph algorithms as well as with a framework for the implementation and integration of new ones.

A convention within *Grph* is that all algorithms are usable through public methods in the `Grph` class. These methods can be straight implementations of algorithms or, better, facility method that call *algorithm objects*. This design may be subject to criticism because it leads to a cumbersome `Grph` class. On the other side, Integrated Development Environments (IDEs) such as Eclipse or Netbeans come with code-completion functionality that all users intuitively use and that make navigation in large source codes easy.

4.1 Defining a new graph property

As detailled in Section, *Grph* graph come with basic property that should suffice for most applications: color, size, shape and label. If these properties do not fit the user need, he may want to introduce new ones.

Most graph libraries provide a specific frameworks for the definition of vertex/edge properties. In some case, these frameworks model properties are *key/value* pairs and they rely on hash-tables to implement them.

```java
class MyGrph extends Grph
{
        AutoGrowingByteArray weights = new AutoGrowingByteArray();

        public byte getEdgeWeight(int e)
        {
                assert getEdges().contains(e);
                return weights.get(e);
        }

        public void setEdgeWeight(int e, byte newWeight)
        {
                assert getEdges().contains(e);
                weights.set(e, newWeight);
        }
}
```

4.2 Defining a new algorithm

In *Grph*, an algorithm is seen as an object.

New algorithms are defined by extending the `grph.AbstractGraphAlgorithm` class. Then you need to provide a name, a textual description, and an implementation for you algorithm. At this step, the newly defined algorithm can be used like:

```java
MyAlgorithm algo = new MyAlgorithm();
algo.compute(graph);
```

4.3 Integrating a new algorithm within *Grph*

In order to properly integrate a newly implemented algorithm within *Grph*, you need to extend the `Grph` class, like in the following example:

```java
public class MyGrph extends Grph
{
        MyAlgorithm myAlgo = new MyAlgorithm();

        public MyAlgoReturnType getMyNewAlgoResult()
        {
                return myAlgo(this);
        }
}
```

If you this the new algorithm is of general interest for the *Grph* community, you may want to propose it to the author so that it will be integrated into the core source code.

Write a facility method in the `Grph` class that calls the algorithm object, following this pattern:

```java
class MyGrph extends Grph
{
        private MyAlgo myAlgo = new MyAlgo();

        public int getMyValue()
```

```
	{
		return myAlgo.compute(this);
	}
}
```

4.4 Defining a LP-based graph algorithm

Linear Optimization functionality is brought via the *Jalinopt* interface to linear solvers.

A new LP-based algorithm for *Grph* is defined by implementing the class `LPBasedAlgorithm`. Doing so only consists in defining the linear program corresponding to the graph problem tackled by implementing the medhod `LP getLP()` and `T processResult(Result r)`

4.4.1 Structured LP...

Coming soon.

4.4.2 Forcing the solver

Override method `LPSolver getSolver()`

4.5 Creating instances with specific properties

4.6 Advanced usage: quest to the maximum performance

Grph makes use of a number if strategies to improve it overall performance. This strategies are detailed in this section.

4.6.1 Caching

In order to improve even more the computational efficiency for the previously mentioned algorithms, *Grph* makes use of a *cache*.

- Basically, any given property will not be computed twice if the graph has not changed since then. This *improves* the complexity of graph operations.
- For example, the computation of all-pair shortest paths will return immediately if the diameter was previously computed . This is because both algorithms rely on a BFS that was first computed by the "diameter algorithm".

Depending on the application, performance may dramatically improves.

Practical experiments show that most time is wasted by computing several times the same algorithms on the same graphs, either directly or not.

In order to avoid this, *Grph* provides a built-in mechanism which stores the results of calculations until a change occur in the structure of the graph. Changes include vertex or edge removal or addition. By default, property changes on vertices or edges are not taken into account (because *Grph* has no mean of detecting them).

Consequently, between two changes of the graph structure (it is also likely that the graph is static, e.g. is will never change), the first invocation of a given graph operation will be performed in the theoretical time complexity, while each subsequent one will be performed in constant time — it actually won't be performed at all, since the cache will bypass it.

When a change occurs (event the slightest one), the entire cache is cleared.

If you want you algorithm to take benefit of the caching mechanism, you simply need to wrap it into a `CachingGraphAlgorithm` object. The following code does it for you:

```
class MyGrph extends Grph
{
        private MyAlgo myAlgo = new MyAlgo().cache(this);

        public int getMyValue()
        {
                return myAlgo.compute(this);
        }
}
```

By doing this, the result provided by your algorithm will be cached until a topological change occur in the graph. The benefit of the caching mechanism is higher when the graph is little dynamic and when computations and time-consuming.

4.6.2 On-demand processing

Processing is done only when the result is requested. This avoids useless processing.

4.6.3 Using native code

Most critical algorithms are implemented in C and compiled to the target machine. These algorithms are called *native algorithms*. Like all algorithms in *Grph*, a native algorithm comes in the form of a Java class. A native algorithm additionally come with a C file. When the algorithm is first executed, *Grph* compiles the C source by using optimizations for the local machine. To do this, the gcc compiler must be available on the corresponding computer. If the C code cannot be compiled, an optional alternative 100% pure Java algorithm is executed.

Many people complain about the poor performance of Java implementations. Our experiments show that C implementations are about 5 times faster than equivalent Java ones.

Grph provides a framework for the development of C implementations for graph algorithms. In order to use it, you need to write the C implementation in the form a set of C functions. Then you need to write a Java interface declaring these C functions, in a way similar to C header files. Then you need to extends the `AbstractNativeGraphAlgorithm` class in order to provide the translation of data between Java and C. At runtime, the C implementation will be compiled on the fly using the local C compiler. The resulting native code will be dynamically loaded and executed.

To write the C code, create a new file called `MyNativeAlgorithm.c`, aside to your java source file:

```
interface MyNativeAlgorithmNativeInterface extends Library
{
        int computeSum(int n, int* vertices)
        {
                // C code...
        }
}
```

You need to declare the native function that will be loaded, as shown in the example:

```
interface MyNativeAlgorithmNativeInterface extends Library
{
        int computeSum(int n, int[] vertices);
}
```

To call the C functions from Java, you need to declare the algorithm class:

```
public class MyNativeAlgorithm extends AbstractNativeGraphAlgorithm<Integer, NativeAlgorithmFu
{
        @Override
        protected Integer nativeCaller(Grph g)
        {
                ((MyNativeAlgorithmNativeInterface) getNativeInterface()).computeSum(list.lengt
        }

        @Override
        protected Class<NativeAlgorithmFunctions> getNativeFunctionsDescriptor()
        {
                return MyNativeAlgorithmNativeInterface;
        }

        @Override
        public String getName()
        {
                return "test native";
        }

        @Override
        public String getDescription()
        {
                return "test native";
        }
}
```

If the native source code is other than C, then you need to specify how to compile it. For example, the use of C is defined by the single following line:

```
AbstractNativeGraphAlgorithm.compilers.put("c", "gcc  -O3 -o $LIB -shared -static -lc $SRC");
```

The `$LIB` and `$SRC` variable will be replaced by the name of the source file and library file that the compiler will generate.

Better avoid native algorithms.

4.6.4 Data wrapping

In order to avoid the construction new large data structure based on other ones, *Grph* makes an extensive use of wrappers. Wrappers are objects that store nothing but simply redirect call to other data structures.

For example, shortest paths do not store a sequence of vertices. They have a reference to the distance and predecessor matrices that provide the necessary information for the shortest paths. This mechanism makes a shortest path of length 10 weigths 4 bytes instead of 56 bytes, hence 14 times shorter.

4.6.5 Parallelism

Modern computers embed a multi-core processor, enabling threads and processes to run in parallel. In order to take advantage of this parallelism, whenever possible *Grph* split the computation in a number of independant jobs that are executed in parallel, within distinct threads, on the local computer.

Depending on the algorithms, the speed off varies from near-to-zero to n,n being the number of processors/cores available on the computer.

Grph comes with a basic parallelization framework for the execution of a certain class of algorithms: if an algorithm can be decomposed in routines called independently on every vertex, it may be invoked by the following code pattern:

4.6.6 Disabling assertions

The code of *Grph* makes an extensive use to assertions. This tends to improve the quality of the source code. Contrarily to the recommandations, *Grph* uses assertions for method parameters checkings too.

This enables the user to disable the assertions, hence disabling all verifications, making the execution significantly faster.

The disadvantage is that the user must not forget to include the following line before the first call the the `Grph` class, in order to enable error verification during the development process.

```
ClassLoader.getSystemClassLoader().setDefaultAssertionStatus(true)
```

4.6.7 Profiling algorithm implementations

Many researchers in algorithmics are interested in performance issues. In order to facilitate the study of the performance for algorithms, *Grph* provides a set of methods enabling to know the duration spent computing.

The implementations of the algorithms that *Grph* uses are the ones that suits better its data structure.

5 Graph import/export formats

Grph provides a set of bridges which make it interoperable with a number of external tools. This interoperability enables persistence, dynamics, two-dimensional rendering (both static and dynamic), etc. The coming sections detail the interoperability capacities of *Grph*.

5.1 *Grph* native formats

Grph natively defines two graph import/export formats: GrphBinary and GrphText.

5.1.1 GrphText: the text *Grph* Language

GrphText is an extension to the common adjacency list format used by Inet, CAIDA maps, etc. It extends it by offering the ability to describe:

- directed edges;
- directed and undirected hyper-edges (bus);
- half and loose edges (edge with one or no end);
- edge identifiers.

GrphText imposes that vertex and edge identifiers are positive integers. The directivity of an edge is described by the use of the character >. Edge identifiers are optional: if they are not given, *Grph* will assign them itself using the natural order $01, 2, 3, etc.$

The format for an ID, called *id* is then `[0-9]+`. The format for an edge declaration is then:

```
(id:)?  id* >?  id*
```

.

As illustrated in the following examples, the compact syntax of GrphText enables the representation of any graph topology.

The line	*Describes...*
`1 2`	an undirected edge connecting vertices 1 and 2
`2 > 3`	a directed edge from 2 to 3
`1:  2 > 3`	the same, but the edge ID is set to 1
`8`	an isolated vertex with ID 8
`{8}`	an undirected hyper-edge that contains vertex 8
`2:  {8}`	an undirected hyper-edge of ID 2, and that contains vertex 8
`{8 4 5 7}`	an hyper-edge connecting vertices 8, 4, 5 and 7
`{8 4} > {5 7}`	a directed hyper-edge connecting vertices 8, 4 to 5 and 7
`# a comment`	guess what...

In GrphText, blank lines are ignored. If a # is found, it is assumed that the rest of the line is a comment.

5.1.2 GrphBinary: the binary *Grph* Format

GrphBinary is the binary counterpart of GraphText. It provides a compact encoding that make Grph data much more compact than GrphText one.

The number of bytes used for the storage of IDs is computed from the total number of elements. If less than 2^8 elements have to be stored, they one byte will be used to store the IDs. If less than 2^16 elements have to be stored, they two bytes will be used to store the IDs. Otherwise 4 bytes are used.

The GrphBinary format is based on the encoding defined by `java.io.DataOutputStream` and is organized as follows:

1. the name of the graph class;
2. the topology, which is decomposed as follows:
   (a) the greatest vertex and edge numbers;
   (b) the number of isolated vertices, following by all of them;
   (c) the number of undirected simple edges, then for each of them:
      i. its number;
      ii. its two adjacent vertices.
   (d) the number of directed simple edges, then for each of them:
      i. its number;
      ii. its source then its destination.
   (e) the number of undirected hyper edges, then for each of them:
      i. its number;
      ii. the number of incident vertices followed by all of them.
   (f) the number of directed hyper edges, then for each of them:
      i. its number;
      ii. the number of tail vertices followed by all of them.
      iii. the number of head vertices followed by all of them.
3. the number of graph properties then, for all of them:
   (a) its name;
   (b) its property-dependant binary encoding;
4. the number of graph algorithms that use caching then, for all of them:
   (a) its name;
   (b) the Java serialization binary encoding for the cached value.

5.2 Others

5.2.1 Dimacs

5.2.2 LAD

5.2.3 DGS

DGS (Dynamic Graph Stream) propose a language for the description of dynamics in simple graphs, directed or not. Practically a DGS stream (coming from a file, a URL, etc) defines a sequence of steps in which a number of events occur. Possible events include: creation/deletion of a node, creation/deletion of a link, modification of the value for a node/link attribute.

Grph provides DGS drivers which make it possible to:

- animate a graph using the DGS language;
- generate DGS text out of the events which happen on the graph (e.g. manipulating a graph though its API entails the generation of the corresponding DGS instructions).

5.2.4 GraphViz dot

Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. Automatic graph drawing has many important applications in software engineering, database and web design, networking, and in visual interfaces for many other domains.

Graphviz is open source graph visualization software. It takes as input a description of a digraph and is able to ouput a variety of popular graphic formats (both bitmap and vector-based).

Grph comes with a GraphViz driver which allow the user to export the graph as GraphViz text. In the case of dynamic graphs, only snapshots can be generated since GraphViz does not consider dynamics. In this case GraphStream can be used.

5.2.5 DotML

Under progress

5.2.6 GraphML

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
        <graph id="1220081709" edgedefault="directed">
                <node id="100" />
                <node id="98" />
                <node id="99" />
                <node id="97" />
                <edge source="97" destination="98" />
                <edge source="97" destination="99" />
                <edge source="98" destination="100" />
        </graph>
</graphml>
```

5.2.7 GML

http://www.infosun.fmi.uni-passau.de/Graphlet/GML/

5.3 Sharing instances

If you wish to share an instance of a given graph, you can store it to a file on disk and send this file via e-mail. Another solution consists in uploading the graph using the `Grph.post()` method. This method will return an URL that you must give to the people you want to send the graph to. These users will be able to load the instance via the `Grph.loadOnlineGrph(url)` method.

5.4 Monitoring

For monitoring purpose, it relies on the simple yet powerful Graphstream.

GraphStream is an open source library which allows the two-dimensional rendering of dynamic graphs. GraphStream listens to graph changes (node/edge creation/deletion and node/edge attribute value modification) and automatically updates the graphical representation. It uses an algorithm providing automatic balancing.

6 Installation

If you plan to use *Grph* through its API, then you need to download the JAR files on *Grph* website and add them to your CLASSPATH.

If you plan to use the experimentation console then you need to install *Grph* on your computer by issueing the following command at your shell promt:

```
curl -s http://www-sop.inria.fr/members/Luc.Hogie/java4unix/j4uni
                             | sh -s grph
```

This command downloads the last version of *Grph* and install it in the `$HOME/.grph` directory. The commands will be copied to the directory `$HOME/.grph/bin` and the jar files will be in `$HOME/.grph/lib`.

7 Third-party software

In order to achieve these goals, it relies on the following third-party software:

GraphStream is a dynamic graph visualization toolkit.

HPPC is a framework for efficient manipulation of Java primitive types.

BeanShell

```
~$ grph-console
Grph - the unpronouceable Graph library. Version 0.9
Copyright CNRS/INRIA/I3S/UNS. 2008-2011.

Grph> g = new Grph()
0 vertices, 0 edges

Grph> g.grid(10, 10)
100 vertices, 180 edges

Grph> g.getVertices().size()
100

Grph> g.getDiameter()
18

Grph> exit
Goodbye.
```

Figure 1: An example code that computes the diameter of a grid of 100 vertices

Java4unix

JLine

8 User interaction

8.1 Reporting

```
~$ grph-console
        Grph g = new Grph();
        g.createVertices(30);
        g.glp();

        Report report = new Report(g);
        RegularFile pdfFile = report.computePDFReport();
```

8.2 Using *Grph* through its interactive console

Grph provides an interactive console like those that use Python and Perl programmers. The console consists in a bridge between the user and *Grph*, meaning that the graph routines remain implemented in Java (they are not reinterpreted by the console). This ensure maximum performance.

The *Grph* Python console can be invoked via the command: `grph-console`.

This console is based on Beanshell. BeanShell is a small, free, embeddable Java source interpreter with object scripting language features, written in Java. BeanShell dynamically executes standard Java syntax and extends it with common scripting conveniences such as loose types, commands, and method closures like those in Perl and JavaScript. In short, BeanShell is dynamically interpreted Java.

```
Grph> new Grph().grid(20, 20).getDiameter()
18
```

Figure 2: The same, but minimal.

8.3 Using *Grph* as a framework

Most Java frameworks suffer exposing their complex architecture to the user. Basic functionalities are often not accessible easily. In the specific case of graph library, most of the time functionalities like creating a graph with a desired topology or obtaining a shortest path require long coding. *Grph* wants to avoid this.

Most of the *Grph* functionality is accessible directly via the `grph.Grph` class.

8.3.1 Creating an ring of 50 vertices

```
Grph g = new Grph().ring(50);
```

8.3.2 Computing all-pair shortest paths

```
g.getAllPairShortestPaths();
```

8.3.3 Getting a particular path

```
Path p = g.getShortestPath(src, dest);
```

8.4 *Grph* UNIX commands

Grph comes with the following command-line utilities. All utilities that accept a graph description as input allow the user to enable/disable parallel computation and result caching.

8.4.1 `grph-compute-properties`

Computes the properties for the given graph. Optionally print their value and profile the algorithm used for computing them.The properties computed can be filtered by a regular expression.

8.4.2 `grph-convert`

Convert the given graph description file to another file.The output format is guessed by the extension of the output file, given as the second argument.If no such 2nd argument is passed, then the conversion is done to all output format supported.These include DHDL, DHDF, GML, GraphML, Inet, etc.

8.4.3 `grph-algorithms`

Print the list of algorithms available in the library

8.4.4 `grph-render-graphstream`

Render the given graph using Graphstream. Since Graphstream does not support hyper-edges, half-edgesand loose edges, these will not be rendered properly.

8.4.5 `grph-illustrate-topologies`

8.5 *Grph* in Mascotte/COMRED

Grph is a Java graph toolkit. It addresses the computational issues encountered by several Research topics at Coati, and to a larger extent at COMRED, which require the programmatic manipulation of large dynamic graphs. In particular, the *study of Internet-like graph properties* requires efficient algorithms for the generation of topologies and for the computation of graph metrics. In addition to this, the *simulation of routing models for the Internet backbone* requires the instantiation of large topologies which do not fit in memory when using common data structures.

A number of graph Java toolkits are already available. These toolkits include JUNG, JGraphT, Mascopt, etc. Each of these projects is designed to a specific purpose. Unfortunately none of them aims at performance and suits our specific needs. To enable experimentation on large graphs (in the order of millions nodes), and to be useful in the context of Coati Research on networks, *Grph* resorts to a number of strategies, not found in other graph toolkits, including:

- its graph models supports directed and undirected simple and hyper-edges in a mixed fashion;
- its data model represents vertices and edges as positive integers (memory-consuming object-oriented design is avoided here);
- its data structure is based on incidence lists, fastening the access to the graph structure (adjacencies, properties, etc);
- it makes use of the parallelism inherent to multi-core processors and multi-processor computers whenever possible;
- the results of graph algorithms are cached as long as they are valid, thereby avoiding lots of re-computation (this is particularly useful in the context of network simulation);
- it uses a compact storage scheme for the serialisation/persistence of graph topology and properties values;
- it provides a console-based interactive interface (shell), making experimentation more accessible.

The design philosophy of *Grph* is to keep its code efficient, and easy to use/understand. The development of *Grph* is driven by the requirements from its users — currently researchers and Ph.D students at Coati and AOSTE. We plan to publicise it for large-scale diffusion.

Grph is registered to APP. It currently has an INRIA proprietary license whose the main idea is that extern development on it have to be made in collaboration with us. See its website for more information: http://www-sop.inria.fr/members/Luc.Hogie/grph/

References

[1] Antoine Dutot, Frédéric Guinand, Damien Olivier, and Yoann Pigné. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. In *EPNACS: Emergent Properties in Natural and Artificial Complex Systems*, 2007.

[2] f. f. http://www.jgrapht.org/.

[3] f. Open-source mathematics software. http://www.sagemath.org/fr/.

[4] Jean-François Lalande, Michel Syska, and Yann Verhoeven. Mascopt - A Network Optimization Library: Graph Manipulation. Rapport Technique RT-0293, INRIA, 2004.

[5] Joshua O'Madadhain, Danyel Fisher, and Tom Nelson. Java universal network/graph framework. http://jung.sourceforge.net/.

[6] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison Wesley, 2002.